VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Porovnání virtuálních strojů Java

Java Virtual Machines Comparison

2017 Tomáš Buchta

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Buchta**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Porovnání virtuálních strojů Java
Java Virtual Machines Comparison

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem závěrečné práce je otestovat možnosti vybraných Java virtuálních strojů nebo nativních kompilátorů: Oracle, IBM, Avian, Graal, JOP, JC, další.
Kritériem pro porovnání bude podpora Java aplikací pro verzi 1.8 a výkonnost. Pro testování výkonnosti budou použity známé testy pro měření výkonnosti jazyků, ale také vlastní vytvořená J2EE aplikace. U této aplikace bude měřena celková doba odezvy, ale také rychlost v jednotlivých částech.

Postup pro vypracování:
1.Zvolte kriteria, na základě kterých budete vybírat zástupce virtuálních strojů nebo kompilátorů.
2. Vyberte zástupce virtuálních strojů nebo nativních kompilátorů pro porovnání.
3. Vyberte sadu testů pro otestování výkonnosti.
4. Navrhněte a vytvořte testovací aplikaci.
5. Proveďte testování výkonnosti.
6. Proveďte porovnání podporovaných vlastností.

Seznam doporučené odborné literatury:

[1] LINDHOLM, Tim, 2014. The Java Virtual Machine Specification, Java SE 8 Edition. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional. ISBN 978-0-13-390590-8.
[2] SETH, Sachin, 2013. Understanding Java Virtual Machine. B.m.: Alpha Science Intl Ltd. ISBN 978-1-84265-815-4.
[3] ANON., nedatováno. The Computer Language Benchmarks Game [online] [vid. 26. únor 2016]. Dostupné z: http://benchmarksgame.alioth.debian.org/
Dále podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2017

.....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 26. dubna 2017 .....

Tímto bych rád poděkoval panu Ing. Janu Kožusznikovi, Ph.D. za odborné vedení a podporu při tvorbě této práce.

Abstrakt

Cílem této práce je srovnání vybraných java virtuálních strojů Javy - HotSpot VM, IBM J9, Graal VM, OpenJDK HotSpot VM, Excelsior JET, Zulu VM. Úvodní část obsahuje specifikaci virtuálního stroje Javy, kde je popsán princip funkce a jednotlivé části virtuálního stroje. V navazující kapitole jsou představeni zástupci virtuálních strojů s popisem jejich klíčových vlastností. V následující kapitole je obsaženo testování výkonosti virtuálních strojů, kde pro každý test existuje graf a tabulka s hodnotami.

Klíčová slova: Java, Java virtual machine, JVM, HotSpot, Benchmark

Abstract

The goal of this thesis is to compare selected java virtual machines - HotSpot VM, IBM J9, Graal VM, OpenJDK HotSpot VM, Excelsior JET, Zulu VM. The introductory part contains a presentatation of the JVM specification, where the function principle and all parts of the virtual machine are described. The next chpter contains introduction and description of the virtual machines and their key features. The performance benchmarks are placed in the last chapter, where each test contains a graph and a table with measured values.

Key Words: Java, Java virtual machine, JVM, HotSpot, Benchmark

Obsah

Seznam použitých zkratek a symbolů

JVM	–	Java Virtual Machine
WORA	–	Write once, run anywhere
JIT	–	Just-in-time
AOT	–	Ahead-of-time
JRE	–	Java runtime enviroment
JDK	–	Java development kit
API	–	Application programming interface
VM	–	Virtual machine
J2EE	–	Java enterprise edition
J2SE	–	Java standard edition

Seznam obrázků

Seznam tabulek

1 Úvod

Programovací jazyk Java vznikl v devadesátých letech minulého století jako řešení problému s přenositelností kódu mezi různými hardwarovými architekturami a operačními systémy. Virtuální stroj je zde v pozici mezivrstvy mezi operačním systémem a samotnou Java aplikací, tím je zajištěno, že Java aplikaci můžeme spustit na jakékoli hardwarové architektuře a operačním systému, pro který existuje implementace virtuálního stroje Javy. Tento koncept ale má i své nevýhody. Využití virtuálního stroje zvyšuje nároky na procesor a paměť počítače a vykonání programu je oproti programům kompilovaným pomalejší. Ve své práci porovnávám jak konkrétní, používané virtuální stroje javy zvládají tyto nevýhody.

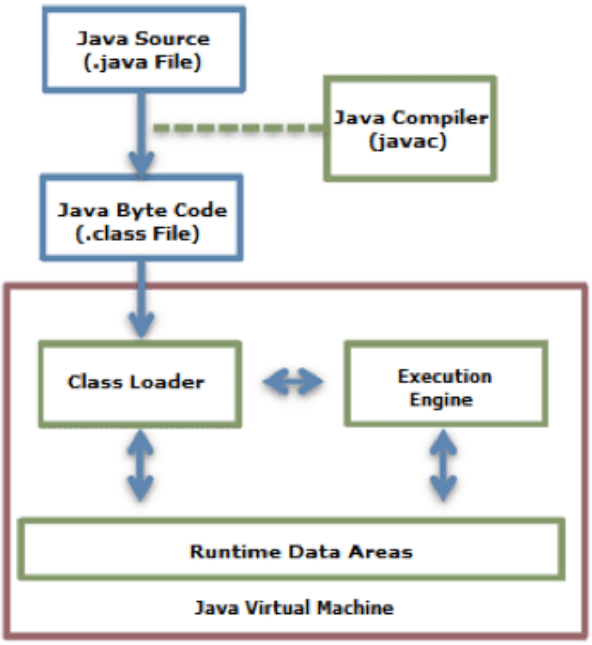
V druhé kapitole se nachází představení a popis specifikace virtuálního stroje javy, který má za úkol nastínit a představit celkový koncept fungování virtuálního stroje i jeho jednotlivé části.

Obsahem třetí kapitoly je představení vybraných zástupců virtuálních strojů javy - Oracle HotSpot, Azul Zulu, IBM J9, Excelsior JET a Graal VM.

Kapitola čtvrtá obsahuje samotné měření výkonnosti jednotlivých virtuálních strojů na testech implementovaných jak pro platformu JavaSE tak pro JavaEE. Každá podkapitola obsahuje jeden test, který je zde představen, grafy a tabulky s naměřenými hodnotami znázorňující výsledek testu a slovní vyhodnocení.

2 Virtuální stroj Javy

Virtuální stroj Javy (**JVM** - Java Virtual Machine) je virtuální stroj definovaný specifikací JVM, který umožňuje spouštění Java aplikací. JVM nepracuje se zdrojovým kódem Javy, ale s tzv. *bajtkód (bytecode)*, který je výsledkem překladu zdrojového kódu kompilátorem a je uložen v *.class* souborech. Tímto je dosaženo tzv. **WORA** (*"Write once, run anywhere"*)[3], Java aplikace může běžet na jakémkoli zařízení vybaveném JVM, nezávisle na hostitelském operačním systému a hardwarové architektuře.[4]



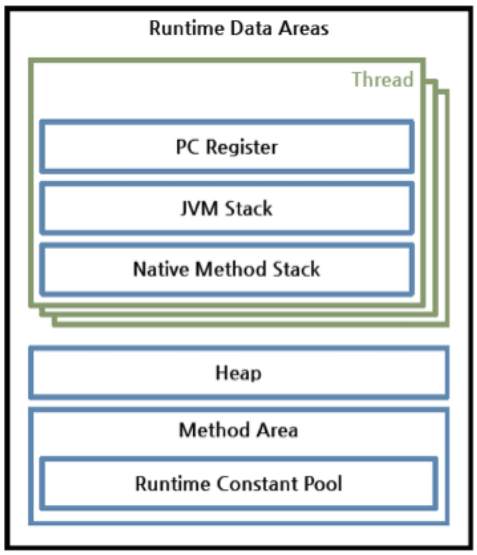
Obrázek 1: Životní cyklus Java programu[5]

Součástí JVM jsou následující součásti

1. Datové oblasti
2. Execution engine
3. Class loader

2.1 Datové oblasti

JVM používá různé datové oblasti, které jsou použity po dobu běhu programu. Datové oblasti můžeme dělit na oblasti, které vznikají a zanikají společně s instancí JVM a jsou přístupné pro všechna vlákna, a na oblasti, které jsou určeny pro každé vlákno zvlášť.[2]



Obrázek 2: Datové oblasti JVM[5]

2.1.1 PC Register

Každé JVM vlákno má svůj pc *(program counter)* registr. Tento registr udržuje informaci o aktuálně prováděné metodě v podobě adresy právě vykonávané JVM instrukce. V případě, že je právě prováděná metoda *nativní*, je hodnota pc registru nedefinovaná. Velikost pc registru je vždy dostatečná pro uložení hodnoty *returnAddress* (pointer na opcode JVM instrukce) na používané platformě. [2]

2.1.2 JVM Stack

Pro každé vlákno existuje jeden JVM zásobník, který vzniká a zaniká společně s vláknem. JVM nemůže přistupovat do zásobníku přímo, využívá proto metod *pop* a *push*. Do zásobníku se ukládají *rámce*. Tyto rámce jsou vytvořeny pokaždé, když je nějaká metoda spuštěna a jsou odstraněny po jejich dokončení. Každý rámec obsahuje ukazatel na pole lokálních proměnných dané metody, zásobník operandů a odkaz na *constant pool* třídy obsahující danou metodu.

Specifikace virtuálního Stroje Java dovoluje autorovi příslušného stroje rozhodnout se, jestli bude mít JVM zásobník pevnou velikost nebo se bude jeho velikost přizpůsobovat aktuální potřebě. JVM zásobníky mohou vyvolat, v souvislosti s pamětí, následující výjimky [2]:

1. *StackOverflowError* - Nastává, pokud vlákno při svém běhu potřebuje větší zásobník, než má k dispozici.

2. *OutOfMemoryError* - Nastává, pokud není dostatek operační paměti pro vytvoření zásobníku pro nové vlákno, nebo je-li JVM nastaveno na automatické přizpůsobování velikosti zásobníku a není volná paměť pro jeho zvětšení.

2.1.3 Native Method Stack

Tyto zásobníky vznikají a zanikají společně s vláknem a jsou využívány pro spouštění *nativních* metod (metody napsané v jiném jazyce než Java, např. v jazyce C). Jestliže implementace JVM nepodporuje práci s nativními metodami (JNI), zásobník pro nativní metody zde nevzniká.

Možné výjimky a jejich způsobení jsou totožné s JVM zásobníky.[2]

2.1.4 Heap

Heap (*halda*) je paměťový prostor společný pro všechna vlákna JVM, zde jsou alokovány instance všech objektů a polí. Velikost haldy může být fixní a nebo může být za běhu rozšiřována (a zmenšována) podle aktuálních požadavků programu. Nevyužívané objekty uložené v haldě jsou odstraňovány a paměť je uvolňována správcem paměti (tzv. *garbage collector*), jehož podoba není JVM specifikací přímo určena. Samotná podoba správce paměti tak závisí na autorovi konkrétního virtuálního stroje.

Například v jazyce C se musí programátor sám postarat o alokaci a následnou dealokaci paměti. JVM si tuto zodpovědnost bere pod svou režii a programátor tak nemusí práci s pamětí řešit.[2]

2.1.5 Method Area

Method Area vzniká při spuštění JVM a je společná pro všechna vlákna. Obsahuje *runtime constant pool*, informace o instančních proměnných a metodách, statických proměnných a *bytecode* metod tříd a rozhraní načtených *Class loaderem*.

Tato oblast je často logickou součástí *haldy* a může být spravována správcem paměti, záleží na autorovi konkrétního virtuálního stroje.[2]

2.1.6 Runtime Constant Pool

Runtime constant pool je reprezentací *constant_pool* tabulek tříd a rozhraní načtených *Class loaderem*. Obsahuje tedy konstanty všech tříd a rozhraní, a reference na metody a proměnné.

Například, je-li v programu odkázáno na nějakou metodu, adresa této metody je získána právě z Runtime constant poolu.[2]

2.2 Execution engine

Execution engine je jádrem každého JVM, jeho hlavní úlohou je vykonávání *bytecode* instrukcí a nativních metod. Každé vlákno JVM využívá vlastní instanci. Specifikace JVM definuje execution engine výčtem instrukcí a popisem jejich významu, samotnou implementaci vykonávání instrukcí ale nechává na autorovi konkrétního virtuálního stroje. Může být využito např. využitím *interpretu*, *just-in-time* kompilace, kombinace více metod a také může být vytvořena metoda zcela nová. [1]

Interpret pracuje s proudem *bytecode* instrukcí, které zpracovává a provádí.

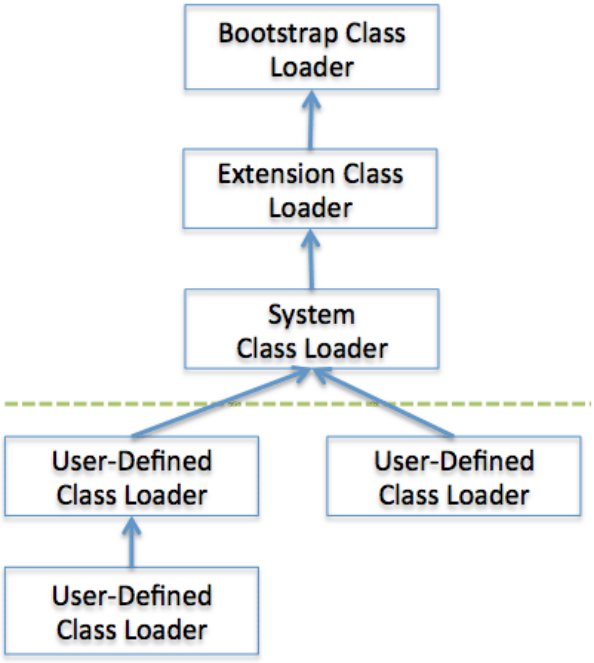
Just-in-time kompilace byla vytvořena pro zrychlení běhu programu. Častokrát opakované části kódu jsou zkompilovány do strojového kódu za běhu programu. Tyto zkompilované části jsou spouštěny rychleji než při použití klasické interpretace.

Každá instrukce se skládá z *opcode* o velikosti 1 byte, následovaným 0 nebo více operandy, nesoucími argumenty nebo data potřebná pro výpočet. Spousta instrukcí žádné operandy nemá a skládá se pouze z *opcode*[1].

Jako příklad instrukce si uvedeme *arraylength*, jejíž opcode je *0xBE*. Instrukce vezme operand typu *reference* s odkazem na pole, jehož velikost chceme zjistit, a uloží velikost na vrchol zásobníku jako hodnotu datového typu *int*[1].

2.3 Class loader

Class loader (zavaděč tříd) načítá *.class* soubory s bytecode a ukládá je do paměti JVM. Nabízí dynamické nahrávání tříd, to znamená, že třídy jsou načítány u příležitosti prvního odkazu na ně, za běhu JVM. Zavaděče tříd mají svou hierarchii (vztah rodič-potomek[1]):



Obrázek 3: Hierarchie zavaděčů tříd.[5]

1. *Bootstrap class loader* je v hierarchii nejvýše postavený. Tento zavaděč je, na rozdíl od ostatních, psaný v nativním kódu. Načítá *Java API*, třídu *Object*, runtime třídy v balíčku **rt.jar**

2. *Extension class loader* načítá třídy v .jar balíčcích *lib/ext* adresáře s *JRE* nebo jiného adresáře definovaného v *java.ext.dirs* (podle hostitelského OS).

3. *System class loader* načítá .class soubory, včetně .jar balíčků, v adresáři definovaném v *java.class.path*, výchozí nastavení je ., aktuální adresář, ve kterém JVM spouštíme.

4. *User-defined class loader* je třídou, která dědí z třídy **java.util.ClassLoader** a je programátorem vytvořena jako jakákoli jiná třída. Vlastní class loader se využívá například pro zavedení specifických bezpečnostních politik při načítání tříd.

Proces načtení třídy začíná požadavkem o tento úkon. JVM načte .class soubor a zkontroluje jeho první 4 bajty. Zde se nachází tzv. *magic number* jehož hodnota je *0xCAFEBABE*. Dále se kontrolují verze .class formátu a další. Poté odpovídající class loader třídu načte do paměti. Následuje *Verifikace* třídy, zde se kontroluje, zda binární reprezentace dané třídy nebo rozhraní má správnou strukturu, zda má každá instrukce správný *opcode*, zda má každá metoda správnou signaturu a podobně. Po této kontrole začíná *příprava* třídy. To zahrnuje alokaci paměti pro statické členy třídy nebo rozhraní a jejich nastavení na výchozí hodnotu. Následuje *incializace*, kdy jsou statické členy nastaveny na požadované hodnoty deklarované programátorem ve zdrojovém kódu. Toto je vykonáno tzv. *statickým inicializátorem*, ten je pro každou třídu unikátní a je reprezentován jménem *<clinit>*. Než může být třída inicializována, musí stejným procesem projít její přímá nadtřída.

Po tomto procesu jsou načtené třídy připraveny k použití a JVM vyhledá metodu *main* v tabulce metod třídy, která byla předána JVM jako první parametr při spuštění.[1, 2]

3 Zástupci virtuálních strojů Javy

3.1 Java HotSpot VM

HotSpot je implementací virtuálního stroje javy původně vyvíjené společností Sun Microsystems pod názvem *Java HotSpot Performance Engine*. Aktuálně je HotSpot vyvíjen a spravován společností Oracle. Jedná se o důležitou součást platformy *Java SE*. Podle údajů o hostitelském stroji je zvolen nejvhodnější kompilátor, nastavení paměťové haldy a správce paměti pro nejlepší výkon.[7]

HotSpot obsahuje dva módy:

1. Server

   Navržen pro maximální rychlost aplikací v serverovém prostředí. Obsahuje složitější adaptivní kompilátor pro nejlepší možnou optimalizaci.

2. Client

   Oproti serveru je navržen pro rychlejší spuštění aplikace a menší paměťovou náročnost. Většina složitějších optimalizací je v tomto módu přeskakována.

3.1.0.1 Paměťový model HotSpot udržuje reference na vytvořené objekty v paměti v podobě ukazatelů na konkrétní adresu v paměti. Tento přístup zajišťuje velmi rychlý přístup k proměnným. Je-li objekt v průběhu kolečního cyklu realokován, je správce paměti zodpovědný za aktualizaci ukazatelů na tento objekt na novou hodnotu.

Hlavičky objektů jsou uloženy jako dvě slova, kde každé slovo má velikost 4 byty ve 32-bitové architektuře a 8 bytů v 64-bitové architektuře. První slovo, označováno jako *mark*, obsahuje informace jako hash objektu a informace pro správce paměti. Druhé slovo, označované jako *klass* (změněné první písmeno, protože v jazyce C++ je "class"klíčové slovo), obsahuje referenci na metadata třídy objektu.[7]

3.1.0.2 Interpret a kompilátor Virtuální stroj HotSpot využívá k interpretaci bajtkódu šablonový interpret. Ten je spuštěn při startu JVM a podle informací v *TemplateTable* jsou vygenerovány šablony bajtkódu pro aktuální platformu.

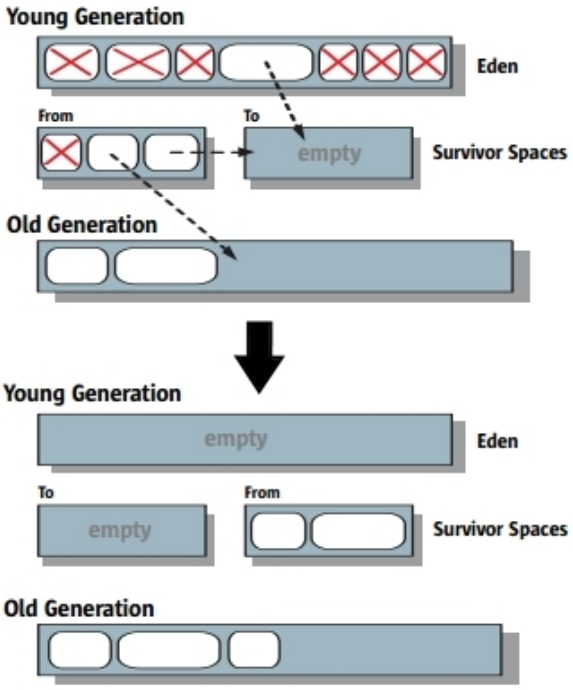
Kompilátor nabízí řadu optimalizačních řešení, např. *Just-in-time* kompilaci (kompilace často volaných částí kódu do nativního kódu), vyhledávání *kritických sekcí* (tzv. Hot spots), eliminace kontroly rozsahu pole, eliminace volání metod (*method inlining*), pro dosáhnutí nejvyššího výkonu.[7]

HotSpot Client VM používá jednoduchý, třífázový kompilátor, navržený pro rychlý běh programu. V první fázi je vytvořena vysoko-úrovňová přechodová reprezentace (*high-level intermediate representation, HIR*) z bajtkódu. HIR využívá *static single assigment (SSA)* formu,

která zajišťuje, že každá proměnná je přiřazena právě jednou a musí být definována před použitím, k reprezentaci hodnot za účelem jednoduššího provedení optimalizací v průběhu a po konstrukci HIR. V druhé fázi je z HIR vytvořena nízko-úrovňová přechodová reprezentace (*low-level intermediate representation, LIR*). Poslední fáze provede alokaci registrů na LIR s použitím upraveného *linear scan* algoritmu, *peephole* optimalizaci a vygeneruje strojový kód. Klientský kompilátor se zaměřuje na kvalitu lokálního kódu a provádí pouze pár globálních optimalizací, kvůli jejich náročnosti na kompilační čas.[7]

HotSpot Server VM je navržen na výkonnost při použití s typickými serverovým aplikacemi. Na rozdíl od klientského VM, provádí server všechny klasické optimalizace včetně eliminace nedosažitelného kódu, vytýkání výpočtů v cyklu a dalších. Alokace registrů je prováděna technikou obarvování grafů, která plně využívá velkých registrů, které jsou v RISC procesorech.[7]

3.1.0.3 Správa paměti Správa paměti v Java HotSpot VM využívá tzv *generační* přístup. Paměť je tedy rozdělena do tří generací: mladá (*young*) generace, stará (*old*) generace a prostor pro metadata (*metaspace*).[6]



Obrázek 4: Paměť před a po kolekčním cyklu.[6]

Mladá generace je rozdělena na tzv. Ráj (*Eden*) a dva přeživší prostory *(*survivor spaces). Nové objekty jsou alokovány v Ráji. Jeden z přeživších prostorů je vždy prázdný a připravený na překopírování prvků, druhý je aktuálně používaný a udržuje prvky, které se zde dostaly z Ráje. V každém kolekčním cyklu v mladé generaci jsou aktivní prvky z Ráje překopírovány do prázdného přeživšího prostoru, aktivní prvky v neprázdném přeživším prostoru jsou také překopírovány do

prázdného přeživšího prostoru - nebo v případě, že přežily určitý počet kolekčních cyklů, jsou povýšeny a přesunuty do **staré generace**. Mrtvé prvky jsou z paměti odstraněny. Kolekční cyklus ve staré generaci se spouští při zaplnění paměti staré generace. V první fázi cyklu jsou prvky označeny (aktivní/mrtvé), v druhé fázi jsou mrtvé prvky smazány.[6]

Prostor pro metadata (dříve permanentní generace) již není součástí haldy. Udržuje metadata pro třídy, rozhraní a zavaděče tříd. Kolekční cyklus je spuštěn, je-li paměť zaplněna.

3.2 Zulu VM

Zulu je open source virtuální stroj, používající OpenJDK, vytvořený společností Azul Systems a zveřejněný v září roku 2013. Společnost také nabízí komerční virtuální stroj Zing. Aktuální verze pro Javu 8 byla vydána v lednu 2016. V současné době je volně ke stažení verze pro OpenJDK 9 s předběžným přístupem.[8]

3.3 Graal VM

Graal VM je virtuální stoj vyvinutý společností Oracle používající technologii virtuálního stroje HotSpot spolu s Graal JIT kompilátorem a JVMCI(*Java virtual machine compiler interface)* JDK. Zaměřuje se na zvýšení výkonu a na podporu více jazyků. Kromě Javy podporuje také JavaScript, Ruby a R. Graal kompilátor je kompletně napsaný v jazyce Java. Vyvíjen je týmem spadajícím pod OpenJDK projekt.[9]

Graal obsahuje 2 interprety:

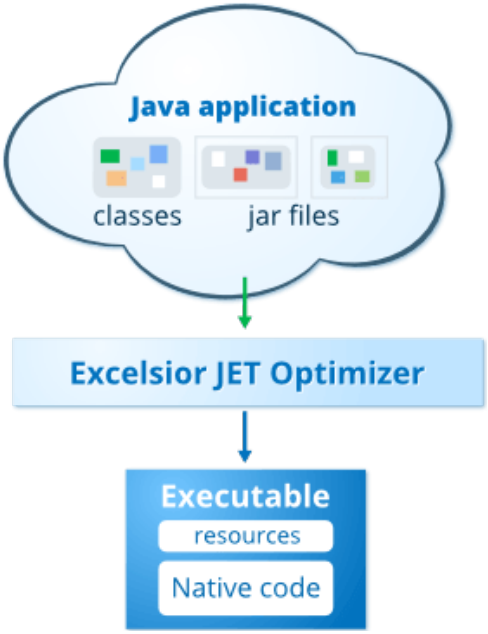
1. Java bytecode interpret.
2. **Truffle** interpret pro **AST**(*abstract syntax tree*)

Součástí virtuálního stroje je tzv. **Truffle API**, které plní funkci Java API, poskytuje AST reprezentaci zdrojového kódu a mechanismus pro konverzi vygenerovaného AST na **Graal IR** (*intermediate representation*).[10]

3.4 Excelsior JET

Excelsior JET je implementace technologie JavaSE postavená na AOT kompilaci Java bajtkódu do optimalizovaného nativního kódu. Autorem je společnost Excelsior LLC, první verze byla vydána v roce 2000 a aktuální(v této práci testovaná) verze 6.3.2017.[11]

3.4.0.1 Excelsior JET Optimizer Optimizer je zodpovědný za statickou ahead-of-time kompilaci bajtkódu. Před prvním spuštěním aplikace je bajtkód optimalizován a zkompilován nativního kódu. Výsledkem je spustitelná aplikace pro platformu Windows, Linux nebo OS X. Po provedení kompilace již nejsou původní .class soubory potřebné pro spuštění, stejně jako není potřebné žádné JRE.[12]



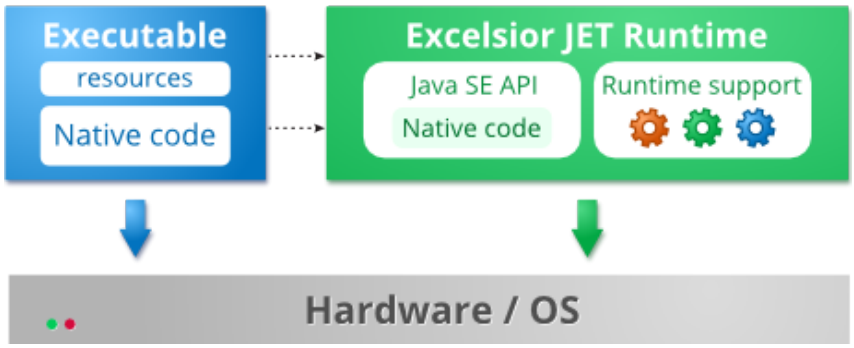
Obrázek 5: Excelsior JET Optimizer - proces. [13]

Kompilací do nativního kódu je dosaženo vysoké bezpečnosti v otázce *reverzního inženýrství*. Oproti jednoduše čitelnému formátu .class souborů je nativní kód daleko složitější na dekompilaci. Tato vlastnost je pro množství vývojářů velkou předností.[13]

3.4.0.2 Excelsior JET Runtime Při spuštění zkompilované aplikace se tato aplikace spojí s běhovým prostředím, tzv. Excelsior JET Runtime, které poskytuje:

1. **Java SE API** - kompletně napsané v nativním kódu.
2. Podporu pro běh programu - Správa paměti (garbage collection)

Tímto odpadá potřeba JVM, aplikace běží rovnou na hostitelském OS.



Obrázek 6: Excelsior JET Runtime.[13]

V důsledku optimalizace a kompilace do nativního kódu ještě před samotným spuštěním aplikace už neprobíhá žádná optimalizace za běhu, to zajišťuje stabilnější běh programu než při použijí klasického JVM řešení.

Zatím co u JVM řešení s JIT kompilátorem se rychlost programu zvětšuje v tzv. *warm-up fázi* (doba, než se provedou veškeré optimalizace), program v nativním kódu žádnou warmp-up fázi nemá a běží tedy s maximální optimalizací ihned po spuštění.

Pro případ, kdy se vývojář rozhodne, že AOT kompilace je pro konkrétní případ nevyhovující, Excelsior JET obsahuje také plnohodnotný JVM(s podporou JIT kompilace) a JRE pro klasické spuštění Java aplikace (*xjava*).

3.5 IBM J9 VM

IBM J9 je virtuální stroj javy vytvořený a udržovaný společností IBM, který je důležitou součástí komerčních softwarových řešení této společnosti, např. aplikační server IBM WebSphere.

J9 používá **sdílení tříd** mezi jednotlivými běžícími JVM instancemi. Třídy jsou při načítání rozděleny na dvě části: část pouze ke čtení (**ROMClass**), kde jsou uložena imutabilní data třídy, a proměnnou část (**RAMClass**), která obsahuje data jako třídní proměnné. Část ROMClass je uložena v sdílené části paměti (*class cache*), která má fixní velikost a do které má každá instance JVM plný přístup. Sdílení tříd vede k úspoře virtuální paměti a také ku zrychlení startu JVM, jelikož se třídy nacházející se v class cache nemusí načítat z disku, ale načítají se z operační paměti.[14]

Další z vlastností tohoto virtuálního stroje je **Ahead-of-time(AOT)** kompilátor, který umožňuje dynamicky generovat nativní kód z bajtkódu za běhu aplikace, jenž je následně uložen do tzv.*shared data cache*. Instance virtuálního stroje následně mohou tento uložený kód nahrát a použít, což je rychlejší, než just-in-time kompilace. AOT kompilátor vybírá metody ke kompilaci s důrazem na zlepšení času spuštění.

J9 umožňuje JIT kompilátoru rozložení úlohy na jednotlivé procesory grafického čipu. Jelikož paralelizace vyžaduje režii, kompilátor ji využije jen tehdy, je li to výhodné. Následující kód je možno vykonat za pomocí grafického čipu[15]

```
public static void main(String[] args) {
   IntStream.range(0, 1000000).parallel().forEach(System.out::println);
}
```

Grafický čip musí podporovat technologii CUDA (*NVIDIA Compute Unified Device Architecture*) a je třeba mít nainstalován CUDA Toolkit 7.5.

4 Porovnání JVM

Veškeré testy byly provedeny ve virtuálním prostředí VirtualBox v následující konfiguraci:

1. Operační systém Linux Ubuntu 16.04 LTS 64bit.
2. 4096 MB vyhrazené operační paměti.
3. 4 jádra procesoru Intel i7-4800MQ@2.70Ghz (100% výkonu jádra).
4. SSD Samsung PM851 256GB.

Kromě virtuálních strojů zmíněných v předchozích kapitolách jsou pro potřeby testování přidány ještě další 2 implementace:

1. Java HotSpot spuštěný s parametrem **-Xint** - interpretovaný mód. V tomto režimu virtuální stroj nepoužívá JIT kompilátor ale pouze interpret. To se projeví na vysokém snížení výpočetního výkonu.
2. **OpenJDK HotSpot VM** - virtuální stroj HotSpot využívající open-source OpenJDK.

Kritérii pro zařazení JVM do testování je podpora Javy verze 8, podpora pro 64bit architekturu (z toho důvodu není porovnáván HotSpot v klientském módu, který na 64 bit architektuře nelze spustit) a primární určení pro desktopové počítače a servery. Pro vyjádření jednotlivých měření grafem je využita hodnota mediánu pro měření každého virtuálního stroje. Osy Y všech grafů vyjadřují hodnoty v nanosekundách.

Všechna měření pro **Excelsior JET** jsou prováděna nad AOT zkompilovaným kódem a všechna nastavení optimalizací jsou nastavena na výchozí hodnoty.

Měření pro **IBM J9** jsou prováděna s parametrem **-Xshareclasses** pro povolení AOT kompilace.

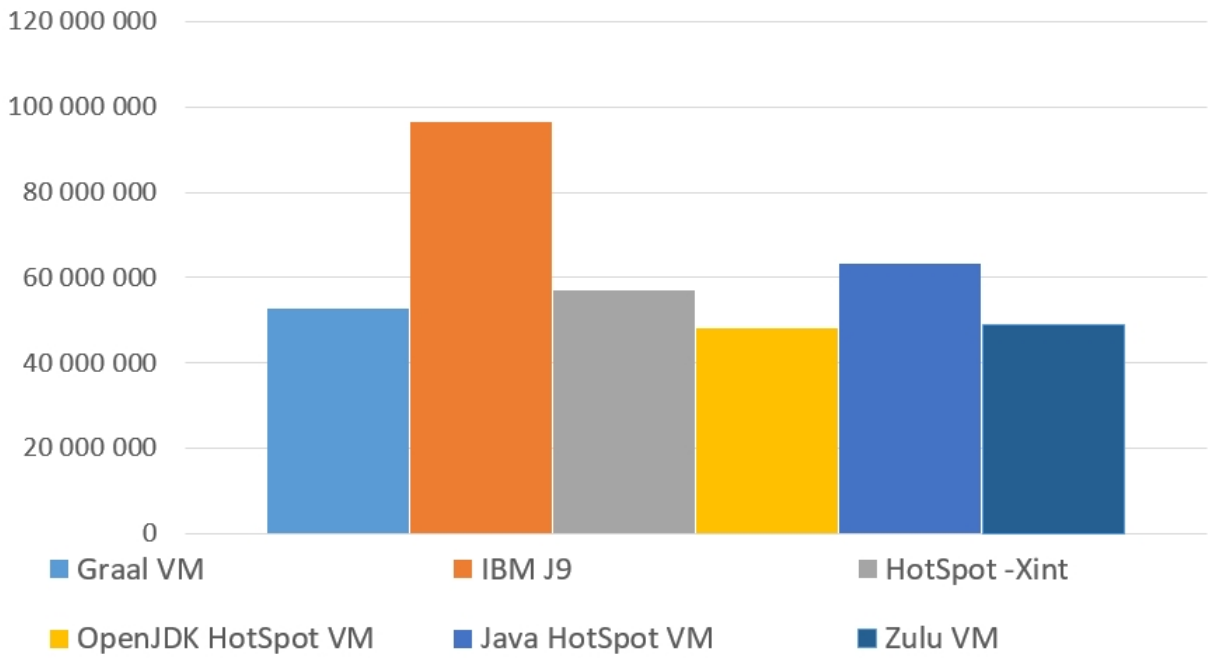
Následující podkapitoly se zabývají jednotlivými testy. Na začátku je uveden popis implementace a poslání testu, následovaný statistickou tabulkou a grafem ke každé verzi testu. V závěrečné části každé podkapitoly je shrnutí a vyhodnocení testu.

4.1 Start JVM

Cílem tohoto testu je porovnat rychlosti startu jednotlivých virtuálních strojů, nejedná se tedy o test výkonu. Měření je prováděno pomocí shellovského skriptu, který v cyklu spouští Java program s prázdnou main metodou, měří se tedy doba od spuštění, do získání návratové hodnoty programu. Technika tedy není zcela přesná, protože je třeba načíst extra .class soubor a také vrátit návratovou hodnotu. Dostačuje ale pro porovnání virtuálních strojů v rovných podmínkách. Start každého JVM je proveden a zaznamenán 100x. Jelikož Excelsior JET při běhu zkompilovaných programů nespouští JVM, není do tohoto testu zařazen.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	52 890 221	60 796 492	29 310 316	302 072 277	46 261 725
IBM J9	96 480 571	103 595 987	17 332 737	159 020 241	88 670 632
OpenJDK HotSpot VM	48 051 250	58 951 582	23 699 652	152 878 388	41 328 930
Java HotSpot VM	63 411 168	75 732 944	26 909 257	168 466 804	47 024 203
HotSpot -Xint	56 893 364	72 109 765	32 140 943	245 896 728	44 284 725
Zulu VM	48 816 977	59 691 597	22 783 040	122 802 640	39 182 468

Tabulka 1: Statistické údaje pro test StartJVM



Obrázek 7: Graf pro test StartJVM

V tomto testu byly nejrychlejší virtuální stroje využívající OpenJDK - OpenJDK HotSpot a Zulu VM naopak spuštění IBM J9 trvalo dobu dvojnásobnou, což je způsobeno vytvářením

sdíleného prostoru pro informace o třídách a zkompilovaný nativní kód. Výsledky měření Java HotSpot virtuálního stroje v obou konfiguracích jsou téměř stejné.

4.2 Test J2EE aplikace

Pro testování J2EE platformy na různých JVM jsem implementoval jednoduchou aplikaci pro provádění CRUD operací nad tabulkami, složenou z datové vrstvy pro přístup k databázi, vrstvy servisní a **REST API** pro zpracovávání HTTP požadavků. Jako běhové prostředí pro tuto aplikaci slouží webový kontejner *Apache Tomcat 7* , který Excelsior JET podporuje pro kompilaci webových Java aplikací do nativního kódu. Pro kompilaci je třeba rozbalit instalační archiv do adresáře a ručně vložit .war soubor s Java aplikací do podadresáře *webapps*. Excelsior JET následně zkompiluje Apache Tomcat včetně cílové aplikace do jednoho spustitelného souboru, který je připraven pro produkci. Všechny konfigurační soubory, které mohou obsahovat citlivé údaje (přihlašovací údaje k databázi, službám) jsou tedy chráněny. Použita je databáze PostgreSQL ve verzi 9.6.2. Pro potřeby měření byla implementována aplikace (Tester.java) která odesílá požadavky na server. Měření bylo prováděno při různých intenzitách odesílaných požadavků:

1. 1 Vlákno odesílající celkem 100 požadavků.
2. 100 Vláken, kde každé vlákno odesílá 100 požadavků.
3. 1000 Vláken, kde každé vlákno odesílá 100 požadavků.

Celé měření probíhalo ve dvou částech:

1. Měření doby kompletního vyřízení celého požadavku v Testeru.
2. Měření doby mezi přijetím požadavku a odesláním odpovědi v J2EE aplikaci.

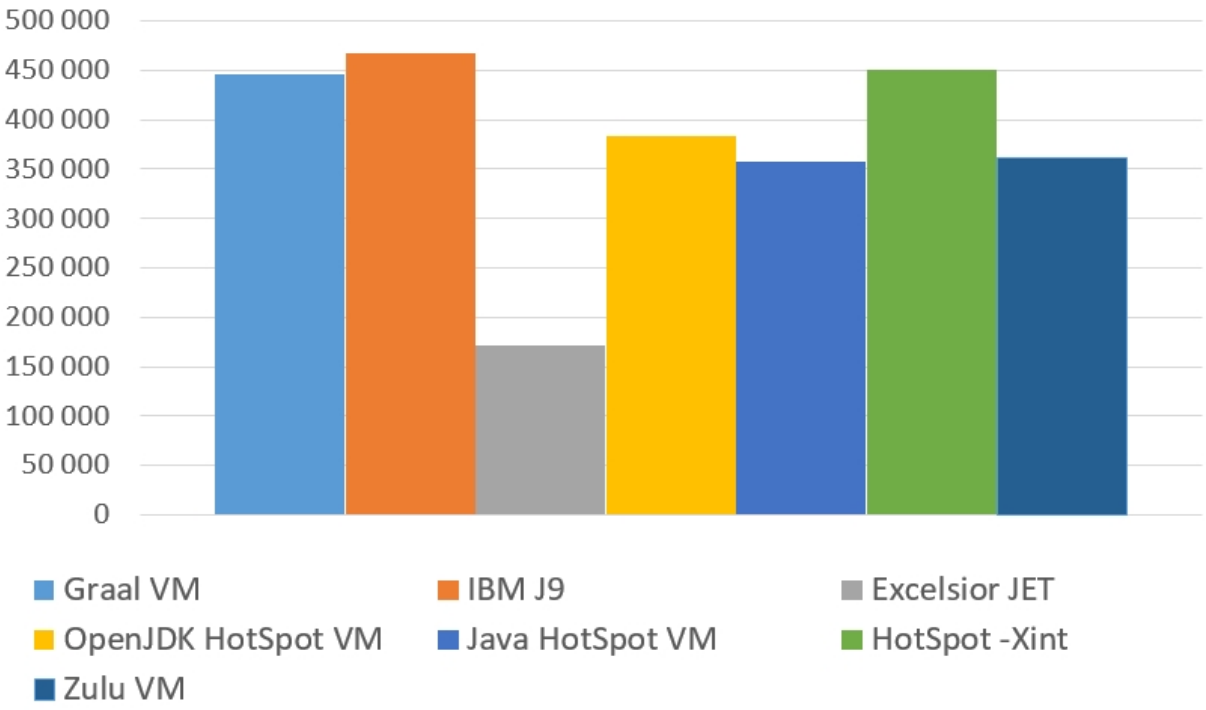
4.2.1 Požadavky z jednoho vlákna

4.2.1.1 Měření uvnitř aplikace

Měření 100 požadavků z jednoho vlákna uvnitř aplikace.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	445 492	1 172 470	4 342 145	43 187 758	294 612
IBM J9	467 627	1 242 981	6 436 122	65 028 758	305 974
Excelsior JET	170 854	266 192	752 625	7 717 736	119 246
OpenJDK HotSpot VM	383 318	1 769 105	12 891 893	129 969 836	271 879
Java HotSpot VM	356 945	1 109 981	5 647 198	56 944 583	226 828
HotSpot -Xint	450 487	1 436 844	9 208 072	93 033 465	292 412
Zulu VM	361 024	1 099 837	6 230 124	62 889 711	266 987

Tabulka 2: Statistické údaje pro test J2EE Inside - 1 vlákno



Obrázek 8: Graf pro test J2EE Inside - 1 vlákno

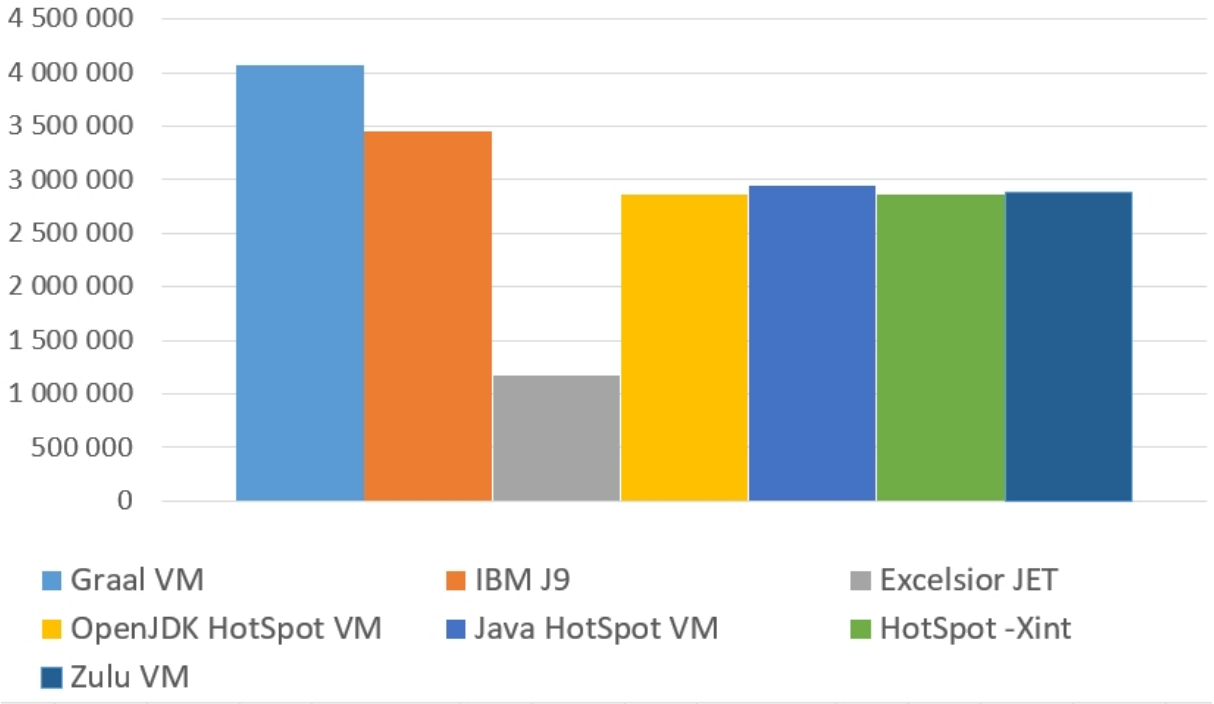
V první části tohoto testu nejlépe dopadl kompilovaný kód Excelsior JETu díky absence warm-up fáze, který má oproti, v pořadí následujícímu, Java HotSpotu a Zulu o více než 100% rychlejší zpracování požadavků. Rozdíl v průměrných hodnotách je ještě větší. HotSpot s vypnutým JIT kompilátorem překvapivě není několikanásobně pomalejší, jako tomu je v ostatních testech ale na JITovaný HotSpot ztrácí přibližně 100 000ns a je podobné úrovni jako Graal VM. Nejdéle trvalo, podle mediánu, zpracování požadavků virtuálnímu stroji J9. Největší kolísání doby vyřízení požadavku při 100 požadavcích vykázal OpenJDK HotSpot.

4.2.1.2 Měření celkové odezvy

Měření kompletního vyřízení 100 požadavků z 1 vlákna.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	4 071 424	7 335 397	15 539 721	152 816 884	1 928 494
IBM J9	3 446 357	6 495 870	19 715 059	199 871 792	1 811 151
Excelsior JET	1 165 025	4 307 056	24 540 252	246 066 769	829 277
OpenJDK HotSpot VM	2 867 202	6 637 536	25 363 217	256 239 298	1 503 007
Java HotSpot VM	2 943 755	6 064 712	18 135 044	182 872 576	1 500 989
HotSpot -Xint	2 859 625	8 324 118	41 125 434	415 686 152	2 069 805
Zulu VM	2 876 593	6 497 117	19 531 490	195 335 770	1 521 976

Tabulka 3: Statistické údaje pro test J2EE Request - 1 vlákno



Obrázek 9: Graf pro test J2EE Request - 1 vlákno

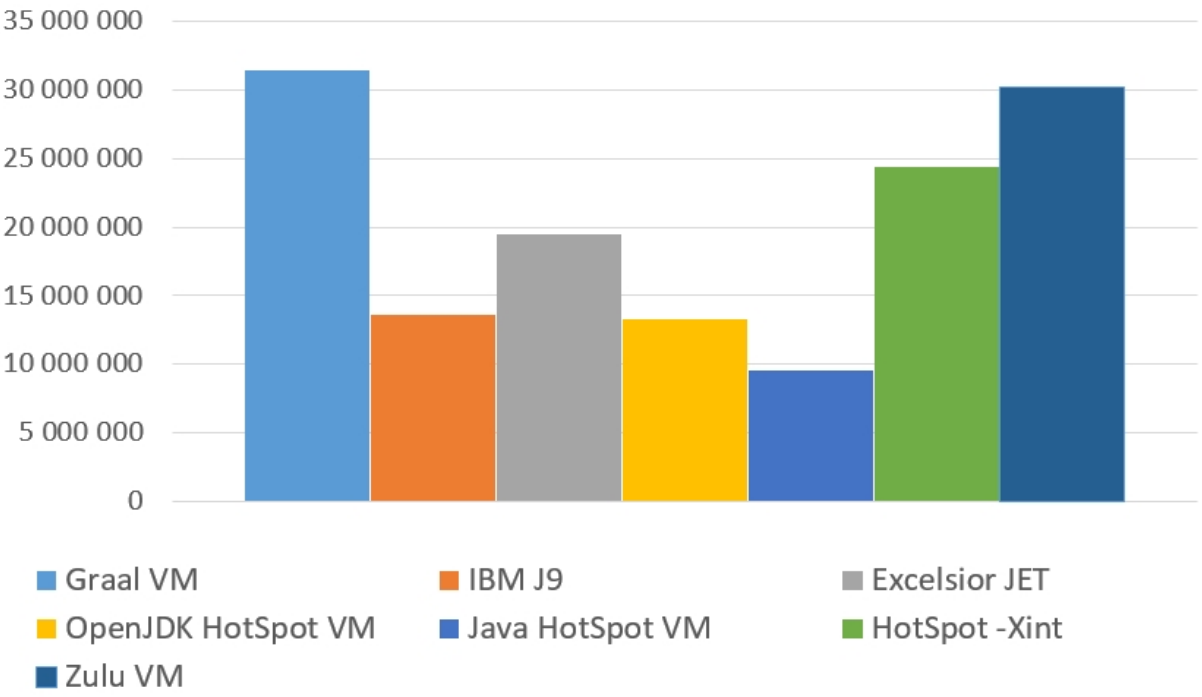
4.2.2 Požadavky z 100 vláken

4.2.2.1 Měření uvnitř aplikace

Měření 100 požadavků z každého ze 100 vláken uvnitř aplikace.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	31 464 894	73 697 818	91 696 254	449 245 824	81 793
IBM J9	13 630 902	25 778 625	34 736 370	526 459 603	119 257
Excelsior JET	19 471 163	24 462 865	25 821 976	292 513 775	93 514
OpenJDK HotSpot VM	13 316 232	35 022 162	45 046 978	221 329 119	80 307
Java HotSpot VM	9 548 114	27 356 355	36 682 744	202 853 214	86 231
HotSpot -Xint	24 414 240	33 152 046	34 504 662	205 787 492	231 168
Zulu VM	30 240 265	66 408 074	81 437 450	387 614 405	87 549

Tabulka 4: Statistické údaje pro test J2EE Inside - 100 vláken



Obrázek 10: Graf pro test J2EE Inside - 100 vláken

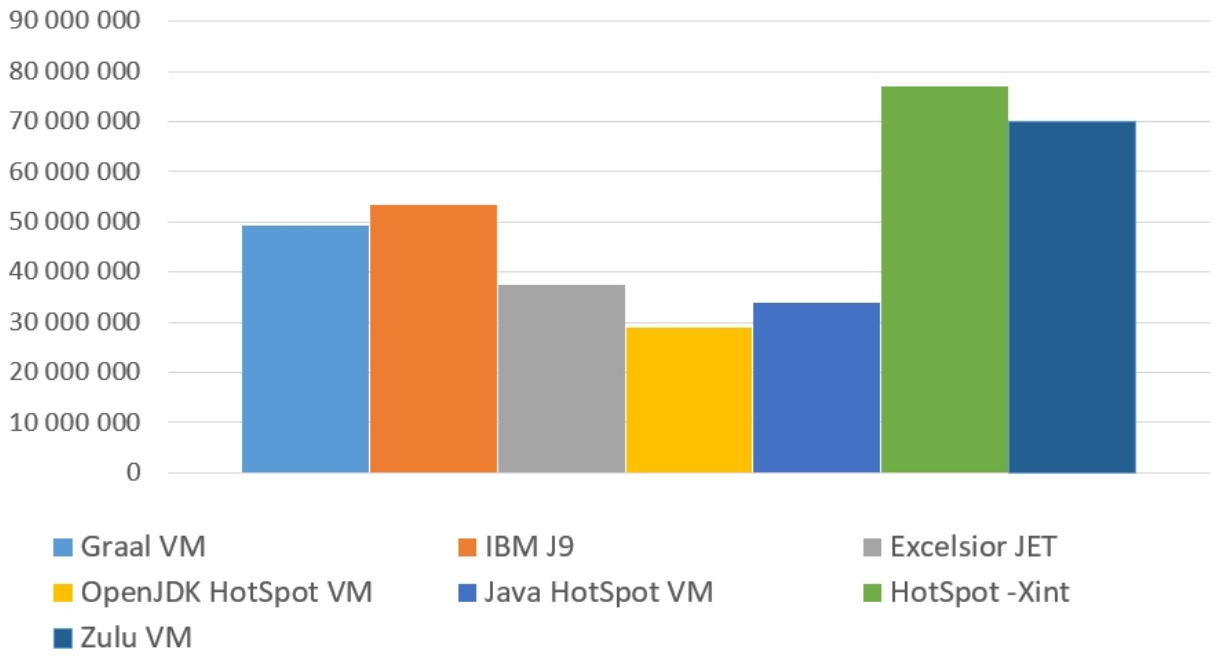
Při celkovém počtu 10 000 požadavků lze pozorovat, že se doba zpracování značně prodloužila. V druhé fázi již Excelsior JET nedominuje, což je zapříčiněno tím, že se u některých virtuálních strojů začíná projevovat JIT kompilaci spojená se zrychlením vykonání. Nejrychlejším JVM v této fázi je Java HotSpot, který přesto, že byl v první fázi stejně rychlý jako Zulu, je teď více než 3 krát rychlejší než Zulu, což je způsobeno dřívějším zásahem JIT kompilátoru. HotSpot s vypnutým JIT kompilátorem se v tomto měření opět umístil překvapivě obstojně. Nejdéle trvalo zpracování požadavku JVM Graal a Zulu.

4.2.2.2 Měření celkové odezvy

Měření kompletního vyřízení 100 požadavků ze 100 vláken.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	49 390 855	88 416 770	95 974 182	465 367 747	385 376
IBM J9	53 399 134	64 817 818	47 350 061	576 231 747	666 649
Excelsior JET	37 550 812	46 539 892	35 420 779	411 182 540	492 017
OpenJDK HotSpot VM	28 846 103	49 564 813	49 688 991	290 934 844	405 162
Java HotSpot VM	33 826 688	49 038 891	43 980 551	373 798 091	468 125
HotSpot -Xint	77 018 957	88 463 580	57 715 558	609 440 361	1 660 174
Zulu VM	70 027 225	101 679 153	87 852 042	451 344 301	446 022

Tabulka 5: Statistické údaje pro test J2EE Request - 100 vláken



Obrázek 11: Graf pro test J2EE Request - 100 vláken

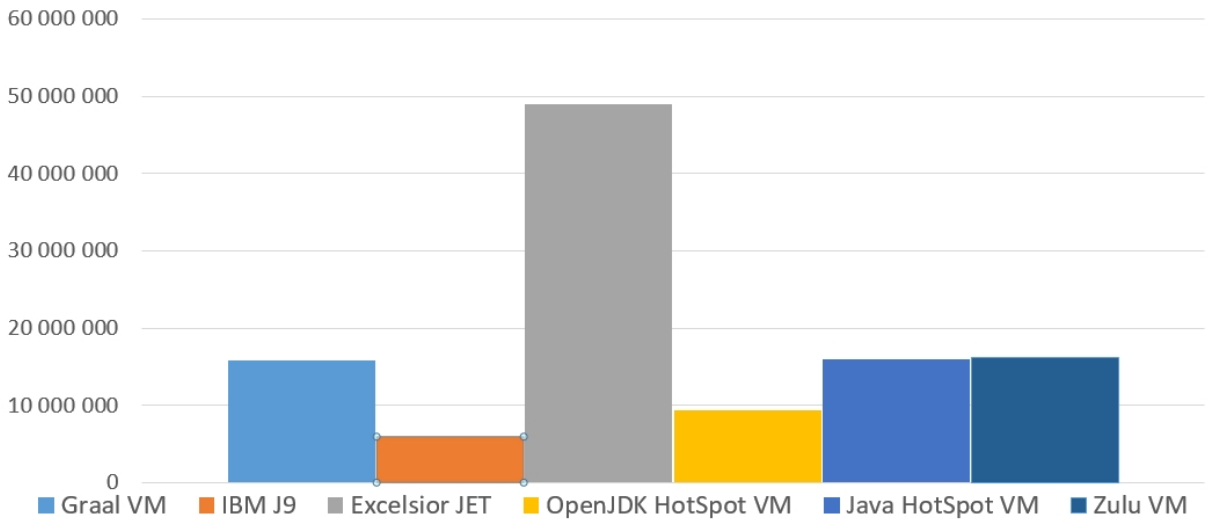
4.2.3 Požadavky z 1000 vláken

4.2.3.1 Měření uvnitř aplikace

Měření 100 požadavků z každého ze 1000 vláken uvnitř aplikace.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	15 799 547	57 334 807	76 253 607	452 206 658	74 716
IBM J9	5 939 096	23 209 866	52 815 201	1 123 308 826	72 257
Excelsior JET	49 018 689	69 1135 999	85 757 742	1 239 447 833	93 140
OpenJDK HotSpotVM	9 374 369	33 844 049	44 913 428	230 034 230	73 048
Java HotSpot VM	16 019 716	43 048 078	54 767 597	404 245 518	75 738
Zulu VM	16 208 876	47 607 321	63 888 783	487 152 352	76 126

Tabulka 6: Statistické údaje pro test J2EE Inside - 1000 vláken



Obrázek 12: Graf pro test J2EE Inside - 1000 vláken

V poslední fázi bylo na server odesláno celkem 100 000 požadavků. V této fázi se již plně projevuje JIT kompilace u JVM, na druhou stranu, kompilovaný kód Excelsior JETu se již ukazuje být méně efektivní a doba zpracování je více než trojnásobná oproti ostatním měřeným virtuálním strojům. Graal, Java HotSpot a Zulu jsou na téměř stejné rychlosti provedení. HotSpot s vypnutým JIT kompilátorem v této fázi měření již nefiguruje, protože v průběhu testu vykazoval výpadky, které způsobily vypršení platnosti spojení s Testerem. IBM J9 měl v této fázi dobu zpracování nejkratší i přes nejvyšší maximální naměřenou hodnotu ze serverů bežících na JVM.

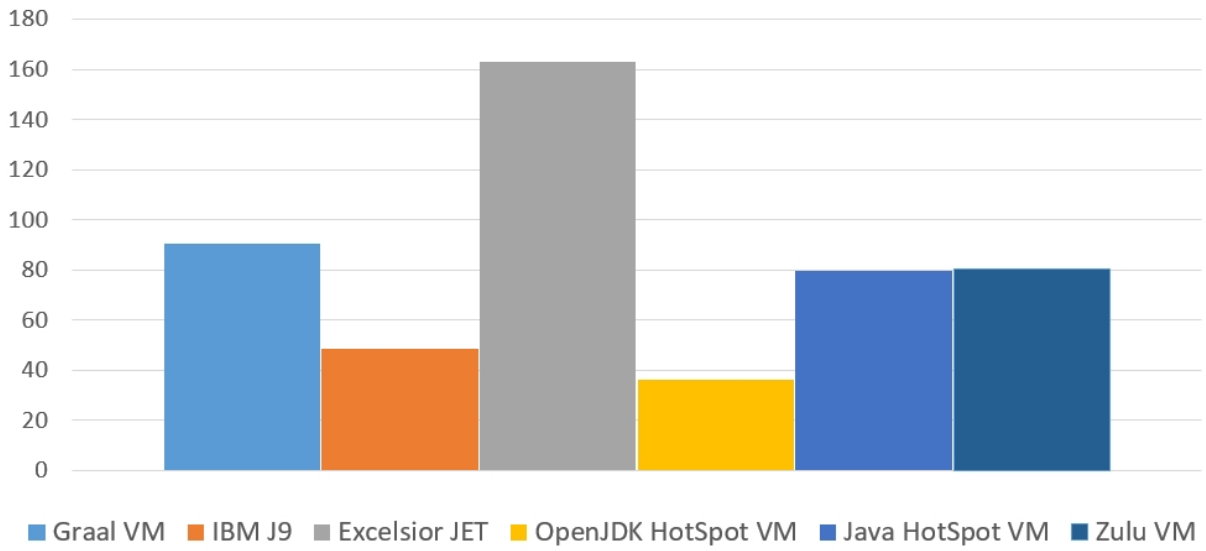
4.2.3.2 Měření celkové odezvy

Měření kompletního vyřízení 100 požadavků ze 1000 vláken.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	90 645 952	294 235 637	767 997 279	8 422 4448 334	461 466
IBM J9	48 498 828	216 899 790	551 453 829	5 106 436 600	503 580
Excelsior JET	163 166 806	522 526 185	1 166 123 078	11 562 719 680	8 310 788
OpenJDK HotSpot VM	36 037 232	127 707 678	418 400 484	4 769 302 249	834 728
Java HotSpot VM	79 730 77	256 601 734	579 137 282	5 928 810 013	1 090 314
Zulu VM	80 462 084	324 985 940	661 517 565	8 268 276 812	447 115

Tabulka 7: Statistické údaje pro test J2EE Request - 1000 vláken

Osa Y tohoto grafu vyjadřuje hodnoty času v jednotkách milisekund.



Obrázek 13: Graf pro test J2EE Request - 1000 vláken

Obrázek 14: Graf pro test J2EE - sumarizace

Při nejnižším měřeném zatížení zpracovával požadavky nejrychleji kód staticky zkompilovaný Excelsior JETem v druhém měření byl nejrychlejší Java HotSpot VM a v nejvyšší měřené zátěži byl nejrychlejší virtuální stroj IBM J9. JIT kompilátor Zulu, J9, Graal a OpenJDK HotSpotu se projevily natolik, že ve třetím testu i přes, oproti druhému testu, zvyšující se zatížení doba vykonání požadavku byla kratší, než v testu druhém, kdežto Excelsior JET, žádne podobné zrychlení neprojevil.

4.3 Test výpočtu faktoriálu

Tento test se zaměřuje na měření rychlosti výpočtu faktoriálu čísla 2000. Měřeno je 1000 po sobě jdoucích výpočtů. Výpočet je proveden dvěma způsoby

1. Rekurzivním voláním funkce.
2. Výpočtem rekurze v cyklu.

Při rekurzivním volání metody je pro každé volání vytvořen rámec obsahující ukazatel na pole lokálních proměnných dané metody, zásobník operandů a odkaz na *constant pool* třídy obsahující danou metodu, který je vložen do zásobníku. Je-li hloubka rekurze větší než jakou velikost má JVM zásobník přidělenou, obdržíme výjimku *StackOverflowError*. Oproti výpočtu pomocí cyklu, je tedy počítat se ztrátou rychlosti z důvodu této režie.

Takto vypadá kód s využitím rekurzivního volání:

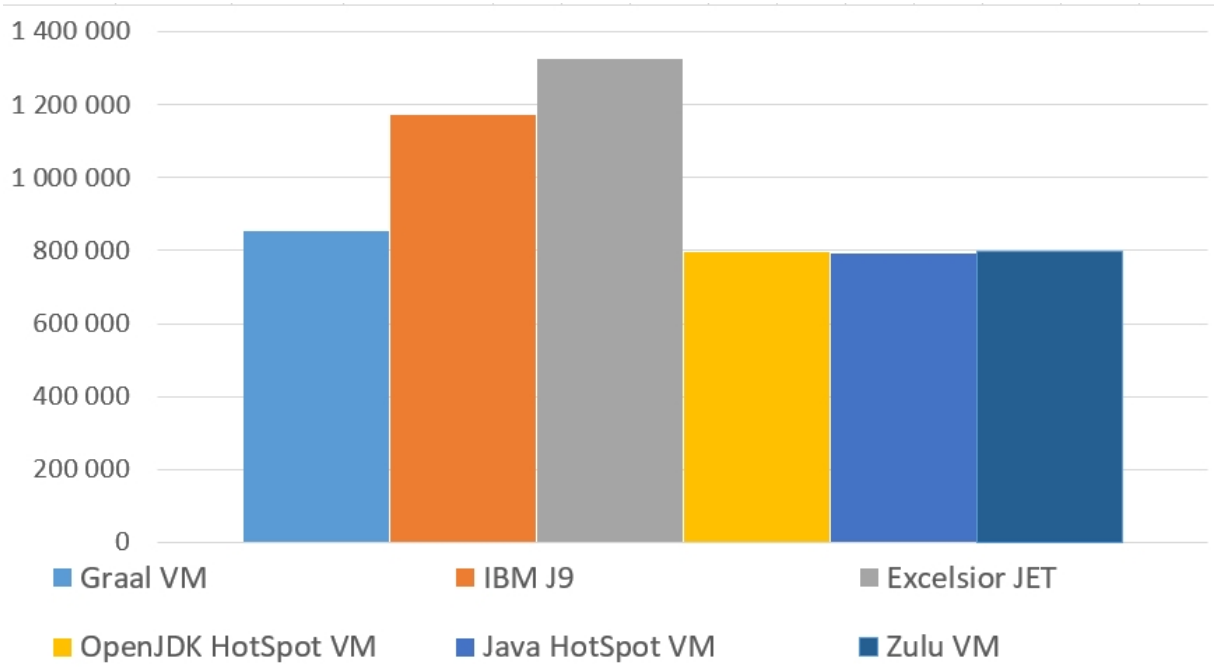
```
public static void main(String[] args) {
        factorial(BigInteger.valueOf(2000));
    }
  }

  public static BigInteger factorial(BigInteger n) {
    if(n == BigInteger.valueOf(1) || n == BigInteger.valueOf(0))
      return BigInteger.valueOf(1);

    return factorial(n.subtract(BigInteger.valueOf(1))).multiply(n);
  }
}
```

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	851 743	1 576 687	2 034 665	47 831 362	739 653
IBM J9	1 172 576	1 396 699	1 678 067	48 940 263	683 383
Excelsior JET	1 325 502	1 809 553	1 156 815	23 334 332	1 205 832
OpenJDK HotSpot VM	795 874	867 382	338 038	6 509 615	693 502
Java HotSpot VM	793 006	871 017	383 445	7 648 671	727 521
HotSpot -Xint	17 779 906	18 143 961	3 747 969	134 283 542	17 253 006
Zulu VM	797 455	887 464	862 553	24 402 411	698 591

Tabulka 8: Statistické údaje pro test FactorialRecursion

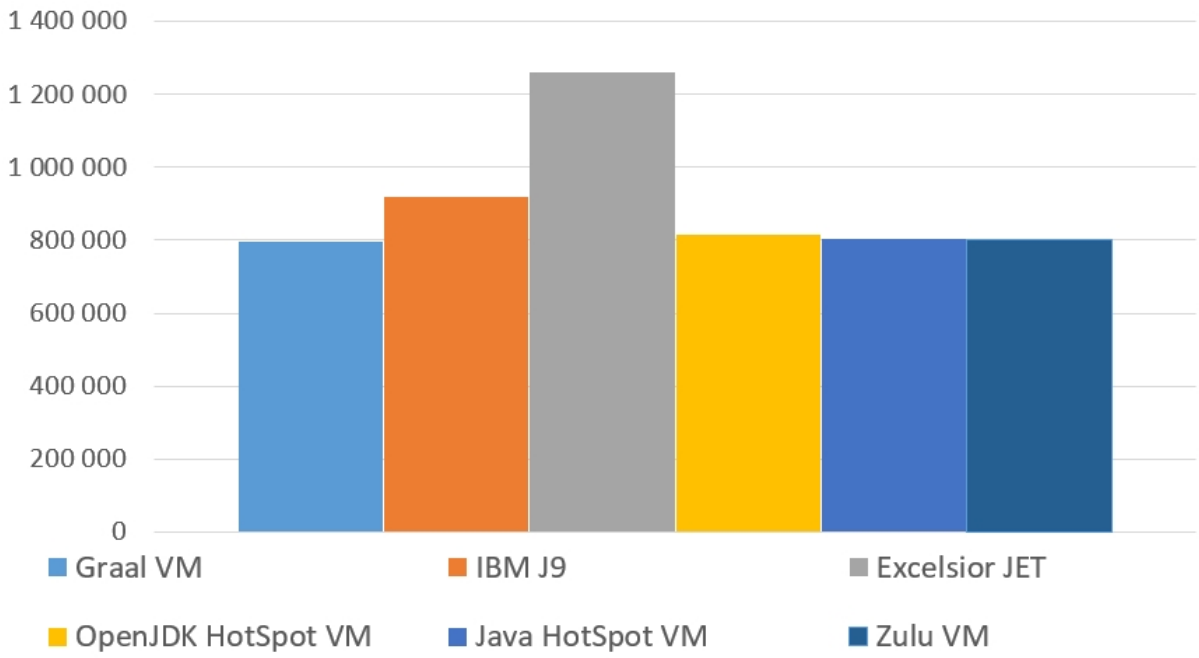


Obrázek 15: Graf pro test FactorialRecursion

HotSpot s vypnutým JIT kompilátorem není v grafu zobrazen, protože rozdíl v rychlosti výpočtu je velmi velký. V tomto testu jsou pro obě verze pořadí jednotlivých virtuálních strojů stejná. Java HotSpot, OpenJDK HotSpot a Zulu mají výsledky téměř identické. Směrodatná odchylka Excelsior JETu, který měl v obou případech nejdelší zpracování, je v rekurzivní verzi dokonce několikrát větší, než odchylka výše zmíněných tří JVM. Rychlost IBM J9 je v tomto testu nižší, než u ostatních virtuálních strojů s dynamickou kompilací kódu.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	797 052	856 773	336 342	5647 171	678 663
IBM J9	919 796	1 073 689	1 587 307	45 421 211	672 360
Excelsior JET	1 258 716	1 435 047	553 432	6 662 975	1 186 828
OpenJDK HotSpot VM	814 399	882 132	403 480	7 878 330	720 216
Java HotSpot VM	803 401	862 452	326 905	6 337 303	676 221
HotSpot -Xint	16 944 955	17 202 743	839 370	29 713 033	16 255 394
Zulu VM	803 171	877 596	440 016	7 453 038	682 939

Tabulka 9: Statistické údaje pro test FactorialLoop



Obrázek 16: Graf pro test FactorialLoop

4.4 Test větvení programu

Účelem toho testu je změřit a porovnat rychlost jednotlivých virtuálních strojů při procházení pole pseudonáhodných celých čísel o velikosti 10 000 prvků, kde pro každé číslo posuzuje, je-li větší než 4. Testy se skládají z 1000 po sobě jdoucích měření. V prvním testu je toto provedeno pomocí konstrukce *if, else if*, v druhém je použit *switch*. Při kompilaci do bajtkódu je v tomto případě switch převeden na tzv. *tableswitch*, který funguje jako pole ukazatelů na pozici v kódu s indexy podle hodnot jednotlivých *case*. Tableswitch má složitost O(1). V případě, že by byly jednotlivé case případy od sebe více vzdáleny, např:

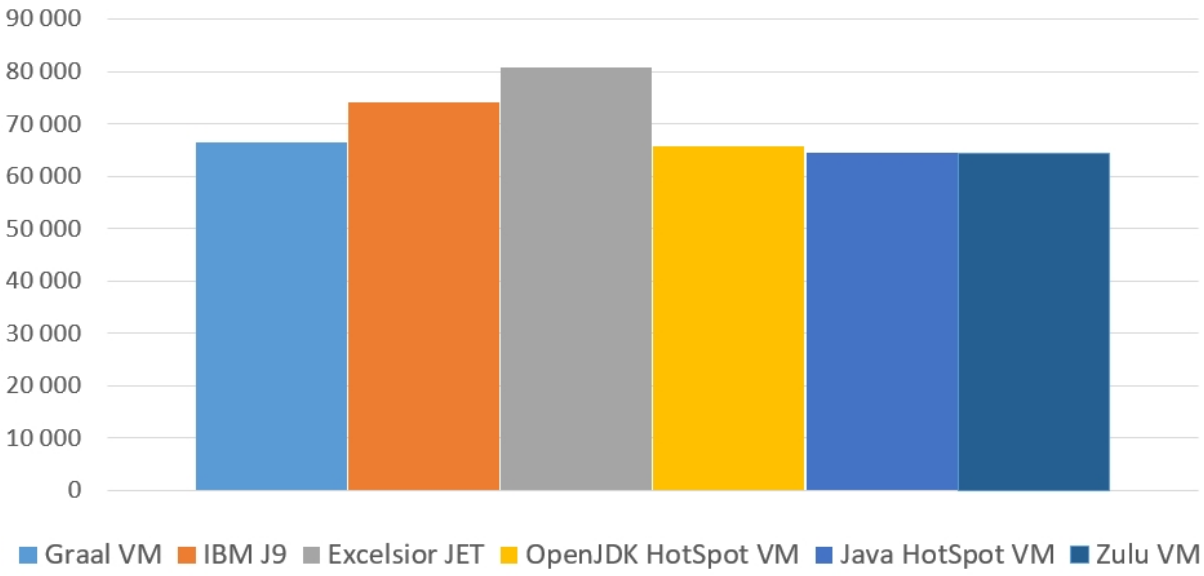
```
switch (inputValue) {
  case 1:   // ...
  case 10:  // ...
  case 100: // ...
  case 1000: // ...
  default:  // ...

}
```

Byl by namísto tableswitch použit tzv. *lookupswitch*. V tomto případě je také vytvořena tabulka s ukazateli na pozici v kódu, ovšem tato tabulka musí být nejdřív seřazena tak, aby platilo že **index** $<$ **index** $+1$. Tato tabulka se poté prochází pomocí binárního vyhledávání, dokud není nalezena příslušná hodnota.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	66 572	218 087	1 345 085	33 502 089	62 837
IBM J9	74 039	100 624	144 915	1 334 567	72 464
Excelsior JET	80 700	85 645	13 518	167 793	77 177
OpenJDK HotSpot VM	65 849	74 331	41 417	612 801	62 987
Java HotSpot VM	64 482	87 052	108 061	3 084 332	61 563
HotSpot -Xint	449 456	488 516	427 495	8 426 886	425 397
Zulu VM	64 397	74 334	43 214	623 844	61 340

Tabulka 10: Statistické údaje pro test Conditions



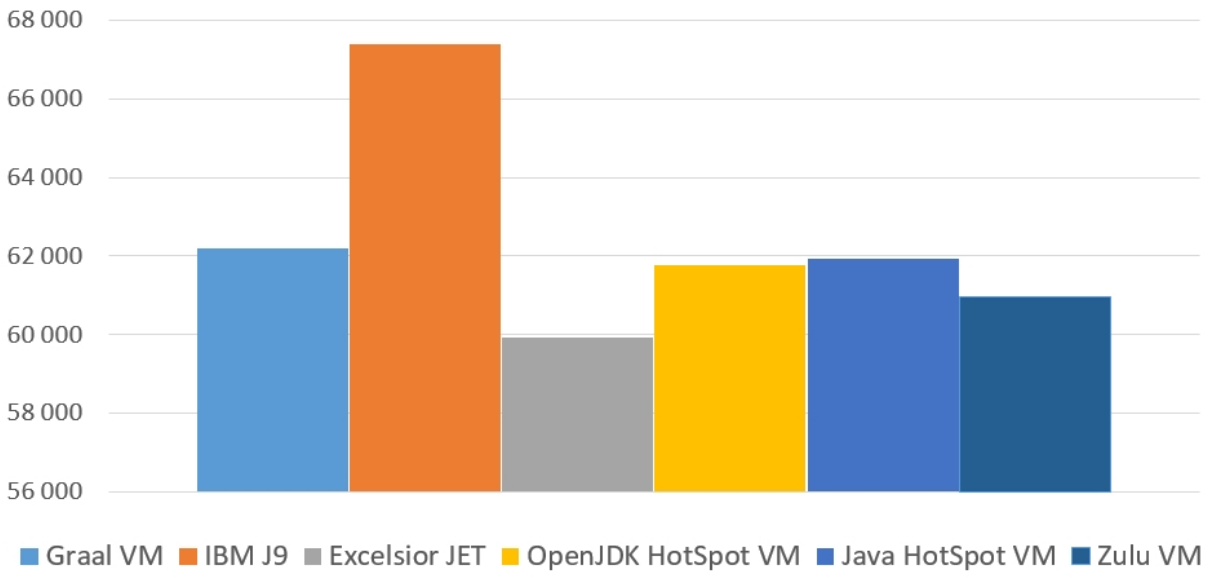
Obrázek 17: Graf pro test Conditions

Test prokázal, že použití konstrukce switch je rychlejší, než identické řešení pomocí if, else if. V prvním měření vykázaly virtuální stroje Graal, OpenJDK HotSpot, Java HotSpot a Zulu velmi podobné výsledky. Nejdéle trval výpočet staticky zkompilovanému kódu Excelsior JETu. HotSpot s vypnutým JIT kompilátorem není v grafu zobrazen kvůli velikému rozdílu hodnot.

Naopak v případě použití konstrukce switch byl Excelsior JET nejrychlejší. I zde jsou výsledky JVM Graal, OpenJDK HotSpot, Java HotSpot a Zulu velice podobné. IBM J9 mělo dobu zpracování největší ze všech porovnávaných virtuálních strojů.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	62 200	70 600	92 410	2 907 431	59 842
IBM J9	67 403	99 757	311 177	8 936 693	64 790
Excelsior JET	59 913	63 965	15 936	434 768	57 658
OpenJDK HotSpot VM	61 779	69 459	85 298	2 669 921	59 852
Java HotSpot VM	61 941	70 597	102 997	3 250 610	59 224
HotSpot -Xint	191 261	199 799	22 143	408 479	185 057
Zulu VM	60 957	68 849	21 224	266 487	59 550

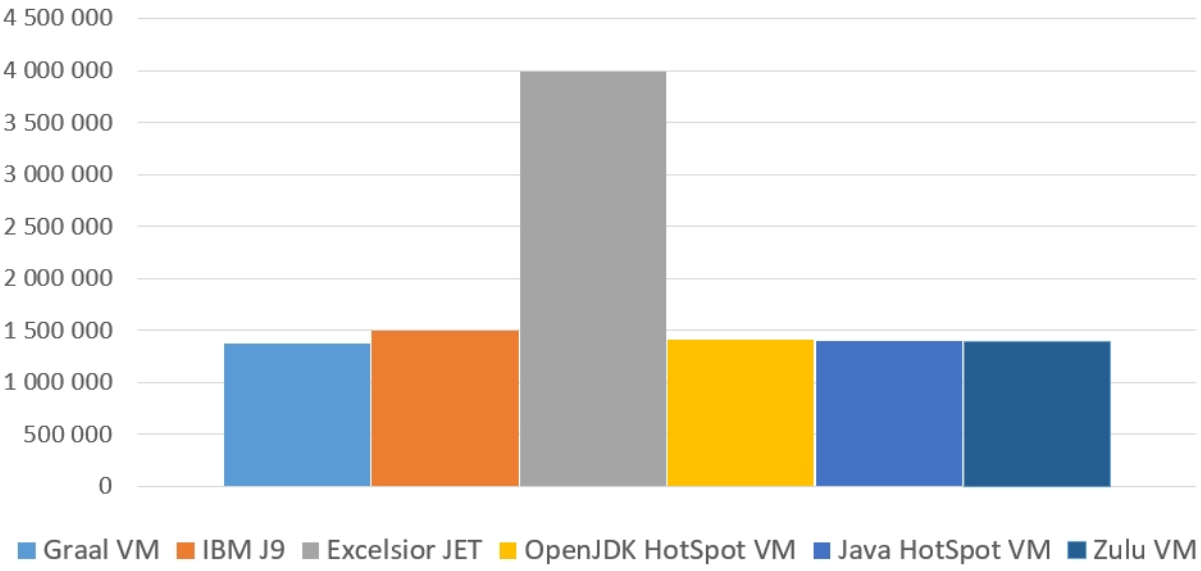
Tabulka 11: Statistické údaje pro test ConditionsSwitch



Obrázek 18: Graf pro test ConditionsSwitch

4.5 Test procházení dvourozměrného pole

Tento test obsahuje měření a porovnání rychlosti jednotlivých virtuálních strojů v otázce procházení dvou rozměrných polí. Pro testovací účely bylo vytvořeno pole o velikosti 2000 x 2000 prvků naplněné pseudonáhodnými čísly v rozmezí 0 - 12345. Testovací aplikace prochází pole prvek po prvku a hledá největší číslo.



Obrázek 19: Graf pro test TwoDimension

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	1 377 262	1 436 775	382 448	7 553 528	1 250 048
IBM J9	1 496 991	1 788 412	1 237 359	32 247 084	1 358 169
Excelsior JET	3 986 621	4 020 102	182 414	6 161 890	3 755 213
OpenJDK HotSpot VM	1 408 258	1 476 691	456 621	9 263 771	1 258 071
Java HotSpot VM	1 394 306	1 471 588	513 204	13 205 901	1 272 00
HotSpot -Xint	55 232 822	55 954 229	3 022 729	94 328 420	53 947 002
Zulu VM	1 398 134	1 497 755	432 688	8 184 874	1 250 480

Tabulka 12: Statistické údaje pro test TwoDimensional

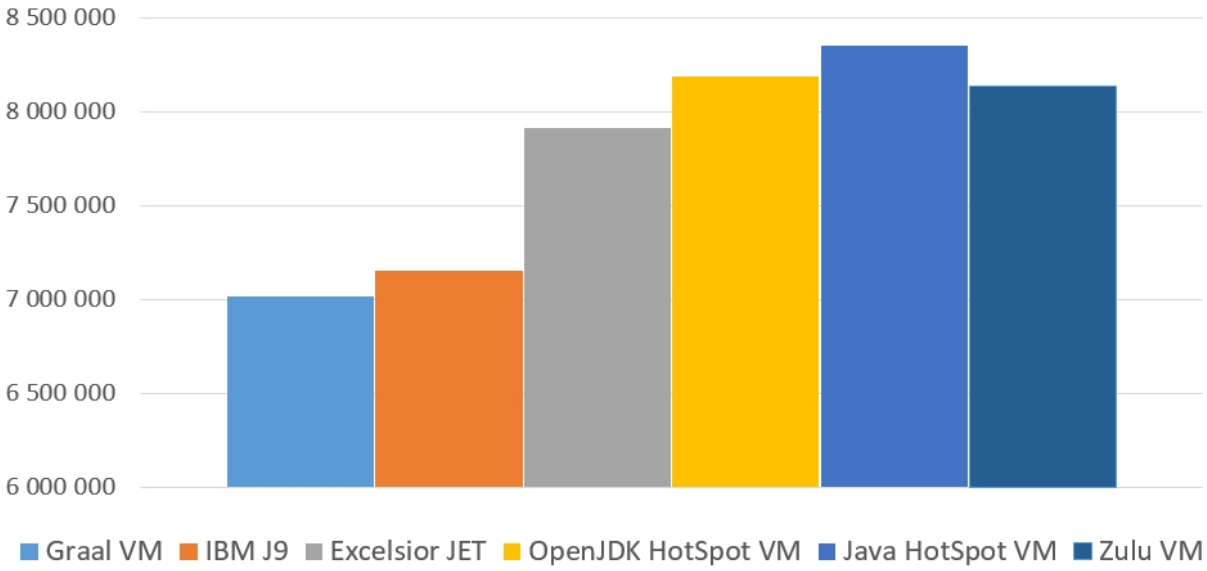
HotSpot s vypnutým JIT kompilátorem není v grafu zobrazen kvůli velikému rozdílu hodnot. Při celkovém počtu 4 000 000 iterací se již plně projevuje rychlost dosažená JIT kompilací, kde Java HotSpot vykonává výpočet více než 39x rychleji. JIT kompilace se projevila u všech virtuálních strojů Javy, proto jsou výsledky velmi podobné. Nejdéle trvaly výpočty staticky zkompilovanému programu Excelsior JETem, kde byla doba výpočtu přibližně 2.5x vyšší než u kódu vykonávaném nad JVM s JIT kompilací.

4.6 Test QuickSort

Quicksort je jeden z nejrychlejších a nejpoužívanějších řadících algoritmů. Jedná se o typ *rozděl a panuj* s průměrnou časovou složitostí O(n log n). Quicksort je ve zlepšené podobě, tzv. *double pivot quicksort* implementován v knihovnách Javy pro použití v metodách *Arrays.sort()*. Tento se test bude tedy hodnotit rychlost seřazení pole o 100 000 prvcích právě tímto algoritmem. Pro zajištění rovných podmínek byla předem vygenerována testovací data o velikosti 100 000 náhodných čísel (quicksort testdata.txt). Tímto se zajistilo, že každý test proběhne nad stejnými daty a tedy algoritmus vykoná stejný počet kroků. Pro každý virtuální stroj bylo provedeno po sobě jdoucích měření.

Virtuální stroj	Median[ns]	Avg[ns]	SD odchylka[ns]	Max[ns]	Min[ns]
Graal VM	7 017 546	7 366 352	1 578 355	21 359 448	6 696 764
IBM J9	7 155 113	7 498 553	4 430 897	143 327 357	6 882 094
Excelsior JET	7 917 032	8 018 789	365 138	11 977 434	7 565 483
OpenJDK HotSpot VM	8 190 078	8 307 015	735 230	21 898 743	7 821 475
Java HotSpot VM	8 354 729	8 613 236	1 450 022	30 935 042	8 014 254
HotSpot -Xint	110 932 462	112 532 825	6 930 934	184 552 275	106 252 946
Zulu VM	8 138 765	8 220 008	492 064	20 788 171	7 796 237

Tabulka 13: Statistické údaje pro test QuickSort



Obrázek 20: Graf pro test QuickSort

V tomto testu provedl výpočty nejrychleji Graal VM, následovaný IBM J9. Staticky kompilovaný kód Excelsior JETu byl ve výpočtu rychlejší, než OpenJDK HotSpot, Java HotSpot i Zulu.

5 Závěr

Práce ukazuje, jak daleko jsou dnešní virtuální stroje javy oproti těm původním v otázce výkonu, díky optimalizačním technikám a Just-in-time a Ahead-of-time kompilaci, viz např. Java HotSpot s vypnutým/zapnutým JIT kompilátorem. Pro potřeby této práce byla navržena a implementována sada testů pro platformu JavaSE, včetně JaveEE aplikace.

Testy ukázaly, že staticky kompilovaných kód Excelsior JETem je prováděn rychleji, než kód spuštěný na virtuálním stroji, v případech krátkých, malých programů. Zde má Excelsior JET výhodu absence potřeby startování virtuálního stroje a absence tzv. warmp-up fáze. Ve složitějších a náročnějších programech již v otázce výkonu zaostává za virtuálními stroji používajícími JIT kompilaci. U JIT kompilaci lze pozorovat proces postupného zrychlování. V rámci testů pro JavuSE se ukazuje, že Java HotSpot a oba virtuální stroje využívající OpenJDK - OpenJDK HotSpot a Zulu jsou si výkonnostně velmi blízko a rozdíly jsou minimální. V J2EE byl z těchto tří nejrychlejší OpenJDK HotSpot. Experimentální Graal byl v některých testech rychlejší a v některých pomalejší, než výše zmíněná trojice. IBM J9 se ze všech JVM ukázal jako virtuální stroj s nejvyššími maximálními hodnotami, což je způsobeno vytvářením a režií sdíleného prostoru mezi jednotlivými instancemi J9. Tyto prostory jsou také velkou výhodou v serverovém prostředí obsluhující větší počet aplikačních serverů (pro IBM jmenovitě aplikační server WebSphere). J9 měl nejpomalejší start, ale nejrychlejší zpracování požadavků v J2EE aplikaci při největším měřeném zatížení. Výkonnost J9 při testech pro J2SE byla pro mě jistým zklamáním. Ukázalo se tedy, že v reálném prostředí je JIT kompilace efektivnější než AOT kompilace a že rozdíly mezi dnešními JVM nejsou drastické.

Pokračování práce by mohlo být zaměřeno na měření Ahead-of-time kompilace, jejíž podpora má být přidána pro Java HotSpot a OpenJDK HotSpot s příchodem Javy 9. Další rozšíření by mohlo být zaměřeno na rostoucí počet počítačových her vyvinutých v jazyce Java (např. Minecraft) a porovnání výkonu virtuálních strojů v této oblasti.

Literatura

[1] SACHIN, Seth. Understanding Java Virtual Machine. Alpha Science Intl, 2013. ISBN 978-1-84265-815-4.

[2] TOM, Lindholm. The Java Virtual Machine Specification, Java SE 8 Edition. Upper Saddle River, 2014. ISBN 978-0-13-390590-8.

[3] Write once, run anywhere. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2017-04-26]. Dostupné z: `https://en.wikipedia.org/wiki/Write_once,_run_anywhere`

[4] Java virtual machine. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2017-04-26]. Dostupné z: `https://en.wikipedia.org/wiki/Java_virtual_machine`

[5] PARK, Se Hoon. *Understanding JVM Internals* [online]. [cit. 2017-04-26]. Dostupné z: `http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals`

[6] SUN MICROSYSTEMS, *Memory Management in the Java HotSpot^{TM} Virtual Machine* [online]. 2006 [cit. 2017-04-26]. Dostupné z: `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`

[7] ORACLE, *The Java HotSpot Performance Engine Architecture* [online]. [cit. 2017-04-26]. Dostupné z: http://www.oracle.com/technetwork/java/whitepaper-135217.html

[8] AZUL SYSTEMS, *Zulu.org* [online]. [cit. 2017-04-26]. Dostupné z: `Zulu.org`

[9] ORACLE, *GraalVM - New JIT Compiler and Polyglot Runtime for the JVM* [online]. [cit. 2017-04-26]. Dostupné z: `http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html`

[10] TOSHEV, Martin. *JVM++: The Graal VM* [online]. 2016 [cit. 2017-04-26]. Dostupné z: `https://www.slideshare.net/martintoshev/jvm-the-graal-vm`

[11] EXCELSIOR LLC, *Product data sheet Excelsior JET*, [Online], [cit. 2017-04-26] Dostupné z: `https://www.excelsior-usa.com/pdf/jet-1130-datasheet.pdf`

[12] EXCELSIOR LLC, *High Performance and Quick Startup*, [Online]. [cit. 2017-04-26] Dostupné z: `https://www.excelsior-usa.com/jetperformance.html`

[13] EXCELSIOR LLC, *How Excelsior JET Works*, [Online]. [cit. 2017-04-26] Dostupné z: `https://www.excelsiorjet.com/internals`

[14] CORRIE, Ben, *Java technology, IBM style - Class sharing* [Online]. [cit. 2017-04-26] Dostupné z: `https://www.ibm.com/developerworks/library/j-ibmjava4/`

[15] IBM, *How the JIT compiler uses a GPU*, [Online]. [cit. 2017-04-26] Dostupné z: https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/understanding/gpu_jit.html

A Přílohy

Obsah CD:

- Tests - obsahuje implementace testů
- J2ee - obsahuje implementace j2ee aplikace
- Results - obsahuje naměřené hodnoty všech testů